> home > about > feedback > login

US Patent & Trademark Office

Try the *new* Portal design

Give us your opinion after using it.

Search Results

Search Results for: **[(replac* or substitut* or chang* or convert* or conversion* or exchang*) <near/5> (false clause* or false predicate* or false statement*)]**
Found **1** of **121,350 searched.**

Search within Results

> Advanced Search

> Search Help/Tips

Sort by: Title Publication Publication Date Score **Binder**

Results 1 - 1 of 1 short listing

1 Variable precision exponential function 97%
T. E. Hull , A. Abrham
ACM Transactions on Mathematical Software (TOMS) September 1986
Volume 12 Issue 2

The exponential function presented here returns a result which differs from e^x by less than one unit in the last place, for any representable value of x which is not too close to values for which e^x would overflow or underflow. (For values of x which are not within this range, an error condition is raised.)

It is a "variable precision" function in that it returns a p-digit approximation for a p-digit argument, for any ...

Results 1 - 1 of 1 short listing

PORTAL
THE ACM DIGITAL LIBRARY

> home > about > feedback > login
US Patent & Trademark Office

Try the *new* Portal design
Give us your opinion after using it.

Search Results

Search Results for: **[(replac* or substitut* or chang* or convert* or conversion* or exchang*) <near/5> (select or "select into" or where or group) <near/5> (clause* or statement*) <near/5> (0 or zero or 1 or one)]**
Found **2** of **121,350 searched.**

Search within Results

GO > Advanced Search
> Search Help/Tips

Sort by: Title Publication Publication Date Score Binder

Results 1 - 2 of 2 short listing

1 On optimizing an SQL-like nested query 99%
Won Kim
ACM Transactions on Database Systems (TODS) September 1982
Volume 7 Issue 3
SQL is a high-level nonprocedural data language which has received wide recognition in relational databases. One of the most interesting features of SQL is the nesting of query blocks to an arbitrary depth. An SQL-like query nested to an arbitrary depth is shown to be composed of five basic types of nesting. Four of them have not been well understood and more work needs to be done to improve their execution efficiency. Algorithms are developed that transform queries involving these basic ty ...

2 Using distributed simulation for distributed application development 99%
Max Muhlhauser
Proceedings of the 21st annual symposium on Simulation January 1988
The software engineering environment DESIGN integrates several approaches for the development of distributed applications. The distributed programming language DC provides for language support. A workstation based human interface integrates programming tools such as a language sensitive editor, a distributed debugger, data evaluation tools, etc. This paper concentrates on a further approach of DESIGN: performance evaluation and prototyping on the basis of distributed simulation

Results 1 - 2 of 2 short listing

Variable Precision Exponential Function

T. E. HULL and A. ABRHAM
University of Toronto

The exponential function presented here returns a result which differs from e^x by less than one unit in the last place, for any representable value of x which is not too close to values for which e^x would overflow or underflow. (For values of x which are not within this range, an error condition is raised.)

It is a "variable precision" function in that it returns a p-digit approximation for a p-digit argument, for any $p > 0$ (p-digit means p-decimal-digit). The program and analysis are valid for all $p > 0$, but current implementations place a restriction on p.

The program is presented in a Pascal-like programming language called Numerical Turing which has special facilities for scientific computing, including precision control, directed roundings, and built-in functions for getting and setting exponents.

Categories and Subject Descriptors: G.1.2 [**Numerical Analysis**]: Approximation—*elementary function approximations*; G.4 [**Mathematics of Computing**]: Mathematical Software—*Algorithm analysis, certification and testing, verification*

General Terms: Algorithms, Verification

Additional Key Words and Phrases: Correctness proof, error analysis, numerical algorithms, variable precision

1. INTRODUCTION

Some programming language extensions that are intended to be particularly helpful in scientific computing have been described elsewhere [10, 11, 13]. The main new feature in these extensions is flexible precision control: the precision of the variables and the precision of the operations performed on the values of variables and constants are declared separately, and both can be changed dynamically.

A consequence of having this flexibility is that, not only do the arithmetic operations and other basic operations (such as *quotient* and *remainder*) have to be carried out in whatever the current precision happens to be, but so do the elementary functions. For example, in the program

```
precision 20
var x, y, z: real
x := 2
y := 3
z := 1 + x + exp(-y)
```

This work was supported by the Natural Sciences and Engineering Research Council of Canada.
Authors' address: Department of Computer Science, University of Toronto, Toronto, Ontario, Canada M5S 1A4.

all operations in the expression $1 + x + \exp(-y)$, including the exponential function operation, have to be carried out in precision 20.

The main purpose of this paper is to describe such an exponential function and to prove it correct. More specifically, it is assumed that all "real" numbers in precision p, other than the number zero, are represented as floating-point numbers with p-decimal-digit significands in $[.1, 1)$ and exponents in $[-10p, 10p]$, and that arithmetic operations are "properly" rounded (i.e., to nearest even in case of a tie). It is then intended that each elementary function return a result that is in error by less than one unit in the last place (1 ulp) in the precision of the calling environment, for an appropriate range of argument values.

A system that satisfies these specifications (for number representation, arithmetic operations, elementary functions, and precision control) has been implemented [13].[1] The language is known as Numerical Turing, which extends Turing (a Pascal-like language developed by Holt and Cordy [9]). The elementary functions for Numerical Turing are given in Abrham's thesis [1], where an earlier version of the exponential function in this paper is presented. (An earlier version of the recently published variable-precision square root function [12] is also presented there.)

The precision control facilities in Numerical Turing make it relatively easy to provide the required accuracy in a straightforward way and to prove the correctness of the result. Two other features of Numerical Turing are also helpful: the *getexp* and *setexp* functions (*getexp*(x) returns the exponent of x, *setexp*(x, n) sets the exponent of x equal to n) and the directed roundings (as indicated later). These facilities are also quite helpful for developing a test program to help verify the correctness of the exponential function.

"Unrestricted" algorithms for the exponential function have been studied from other points of view, for example by Brent [3, 4, 5] and by Clenshaw and Olver [6]; see also Borwein and Borwein [2]. Besides a careful design, analysis, and testing of the program, some emphasis in this paper is placed on programming language aspects.

2. OUTLINE OF PROGRAM

The purpose of the program in Figure 1, given x, is to return an approximation to e^x that is in error by less than 1 ulp. Both x and the returned value $\exp(x)$ are in whatever precision has been specified for the calling environment.

A key part of the strategy for calculating this approximation is to determine an integer k so that (i) $r = x/k$ is exact, and (ii) $|x/k| < 1$ (so that a truncated Taylor series can be used to approximate $e^{x/k}$ reasonably efficiently). The approximation to $e^{x/k}$ is accumulated in *sum* in the program, and then *sum*^{k} provides an approximation to e^x (which, because it is calculated in higher precision, must still be rounded to produce the final value of $\exp(x)$).

The major purpose of the analysis in the next three sections is to determine what precision p to use in evaluating the truncated Taylor series, and to determine how many terms n to take in the series. Before considering the detailed analysis,

[1] This system is available from the first author (for a VAX under 4.2 BSD Unix); the precision is currently restricted to [1, 200].

```
function exp(x : real) : real
    % exp(x) returns an approximation to the exponential of x
    % which differs from the true value by less than 1 ulp, in
    % the precision of the calling environment, or it fails if
    % abs(x) is beyond an over/underflow threshold
    %                    t e hull and a abrham, may 1986

    % first deal with special cases
    if abs(x) > 23*currentprecision then
        assert false                    % fail if abs(x) beyond an
    end if                              %   over/underflow threshold
    if abs(x) <= setexp(.9, -currentprecision) then
        result 1                        % return 1 if abs(x) small
    end if                              %   enough; this also avoids
                                        %   later over/underflows

    % use t to determine k, r and p; t is the smallest
    % integer >= 0 such that the corresponding abs(x/k) < 1
    const t := max(getexp(x), 0)
    const k := 10**t                    % k is the reduction factor
    const r := x/k                      % r is the reduced argument
    const p :=                          % p is precision for
        currentprecision + t + 2        %   calculating sum

    % determine n, the number of terms for calculating sum;
    % use first Newton step, (1.435p - 1.182)/log10(p/abs(r)),
    % for solving appropriate equation, along with directed
    % roundings and simple rational bound for log10(p/abs(r))
    precision max(p, 4)
    const numer := 1.435*p - 1.182  % exact numerator
    const pbyr := divrd(p, abs(r))  % lower bound for p/abs(r)
    const f := setexp(pbyr, 0)      % fraction part of pbyr
    const denom := subrd(getexp(pbyr),
        divru(1-f, addrd(mulrd(1.558,f),.7441))) % lower bound
    const n := ceil(divru(numer, denom))  % upper bound

    % find sum of truncated Taylor series for exponential
    % of r, using Horner's scheme
    precision p
    var sum : real := 1
    for decreasing i : n-1 .. 1
        sum := sum*(r/i) + 1
    end for

    result sum**k
end exp
```

Fig. 1. Variable precision exponential programs.

we describe the steps in the resulting program in more detail. This can be done conveniently in five stages.

Stage 1. Two special cases are dealt with in this stage, one when $|x|$ is large and the other when $|x|$ is small.

To avoid overflow and underflow in the final result, we must have

$$.1 \times 10^{-10cp} \le e^x \le (1 - 10^{-cp}) \times 10^{10cp},$$

where cp is an abbreviation for *currentprecision.* (The Numerical Turing function *currentprecision* returns the current precision of the environment.) This means that

$$-10\text{cp} - 1 \le x \log e \le \log(1 - 10^{-\text{cp}}) + 10\text{cp}.$$

Noting that $1/\log e = \ln 10 = 2.30258\ldots$, it can be verified that it is sufficient to require $|x| \le 23\text{cp}$, for $\text{cp} \ge 2$. This restriction on $|x|$ is also sufficient when cp is 1, since then the 23 is converted to type *real* and rounded to precision 1, which means it is rounded down to 20, or $.2 \times 10^2$ in floating-point representation, before being compared to $|x|$. The first part of stage 1 causes a failure to occur if $|x| > 23 \times$ *currentprecision*, which ensures that x is kept within a range that is just sufficient to avoid overflow and underflow.

In the remaining part of the first stage, the value 1 is returned to the calling program if $|x| \le$ setexp (.9, *−currentprecision*) $= .9 \times 10^{-\text{cp}}$. The error E in this case satisfies

$$\begin{aligned} |E| &= \left| x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots \right| \\ &\le |x| + |x|^2 + |x|^3 + \cdots \\ &= \frac{|x|}{1 - |x|} \le \frac{.9 \times 10^{-\text{cp}}}{1 - .9 \times 10^{-\text{cp}}} \le \frac{.9 \times 10^{-\text{cp}}}{1 - .9 \times .1} < 10^{-\text{cp}}, \end{aligned}$$

since $\text{cp} \ge 1$. We can conclude that $|E|$ is less than 1 ulp.

Thus the required accuracy is easily guaranteed for values of x such that $|x| \le .9 \times 10^{-\text{cp}}$. This bound is close to the largest possible for which such a result can be guaranteed, but the main reason for considering this special case is that it also guarantees that there will be no overflow or zerodivide when *pbyr* is evaluated later in the program.

Stage 2. The value of k and $r = x/k$, as well as the precision p, are obtained in this stage. In order that r be exact, it was decided to make k a power of 10, in fact the smallest nonnegative power of 10 such that $|x/k| < 1$. The appropriate power is denoted by t.

The analysis presented in Section 3 shows that the precision p, where

$$p = \textit{currentprecision} + t + 2$$

is high enough to ensure sufficient accuracy in the evaluation of the truncated Taylor series, as well as in the final calculation of $\textit{sum}^k$. The constant p is therefore evaluated at the end of this stage.

Stage 3. To avoid having to test each new term in the Taylor series to determine when to truncate the series, we need to determine in advance the number of terms to be used. This also makes it possible to evaluate the truncated series according to Horner's scheme, which is more accurate in terms of the cumulative effect of round-off error (and it happens that we can obtain a bound on this error that is independent of the number of terms, as explained in Section 4).

The number of terms n to be used in the Taylor series is determined in this stage. The value of n is not the best possible, or ideal value, but it turns out to be a reasonably close upper bound. The analysis in Section 5 shows how it is obtained. A certain nonlinear inequality involving the truncation error in the Taylor series relates the ideal value of n to p and r. One Newton step applied to a corresponding equality leads to a formula for the value of n. (Directed roundings are used to ensure that the value of n is an upper bound for the ideal value. For example, *divru*(x, y) and *divrd*(x, y) are the rounded up and rounded down quotients of x by y, respectively. Similarly for *add*, *sub*, and *mul*.)

Evaluating a truncated Taylor series, as is done here, requires one additional floating-point operation per term, compared to evaluating a fixed-length polynomial such as a Chebyshev polynomial. And when $|r|$ is near its maximum value, more terms will be needed than with a Chebyshev polynomial. However, there is some compensation in that n depends on r as well as p, so that fewer terms are required when $|r|$ is not near its maximum value.

Stage 4. The truncated Taylor series

$$1 + r + r^2/2! + \cdots + r^{n-1}/(n-1)!$$

is evaluated at this stage, in a form suggested by Horner's rule. The calculations are done in precision p and the result is accumulated in *sum*. The value of *sum* is an approximation to $e^r = e^{x/k}$. (We show at the end of Section 5 that no underflow can occur in these calculations.)

Stage 5. It remains only to raise *sum* to the power k in order to produce an approximation to e^x. This value is in precision $p = \mathit{currentprecision} + t + 2$, and, as explained in the next section, differs from the exact value of e^x by less than 1/2 ulp in the precision of the calling environment.

There is an implicit assignment with the return of this value, and a consequent rounding to the environment precision, which can introduce a further error of at most 1/2 ulp in that environment precision. The final value, which is available at the point of invocation of the function in the calling program, therefore differs from the exact value of e^x by less than 1 ulp.

All of the above assumes that floating-point precision is potentially unlimited and that the integers are potentially unbounded. In a practical implementation, both of these will of course be constrained, and, as a consequence, the program will run correctly only if these constraints are not violated. The relatively minor modifications to the program and its specification that are needed in practice are explained in the last section.

3. DETERMINATION OF PRECISION

To determine the precision p to be used in evaluating *sum*, and in the subsequent raising of *sum* to the power k, we have to analyze the errors incurred.

There are three sources of error. We use E_T to denote the relative truncation error in using the truncated Taylor series as an approximation to e^r, and E_R to denote the relative round-off error that accumulates in the evaluation of that truncated series. We then have

$$\mathit{sum} = e^r(1 + E_T)(1 + E_R).$$

This value is raised to the power k, and (in Numerical Turing) this is equivalent to $k - 1$ multiplications in so far as the error analysis is concerned. It turns out to be convenient to allow for at most k rounding errors, rather than $k - 1$, so that we have

$$sum^{**}k = e^x(1 + E_T)^k(1 + E_R)^k(1 + E_p)^k,$$

where $|E_p| < 5 \times 10^{-p}$ (a relative round-off error in precision p is bounded in magnitude by 5×10^{-p}).

We want the value of $sum^{**}k$, before the rounding that is implicit in returning this value to the calling environment, to differ from the exact value of e^x by less than 1/2 ulp in the precision of that environment. It is sufficient to require that

$$|sum^{**}k - e^x| < .5 \times 10^{-\mathrm{cp}}e^x,$$

where cp denotes the precision of the calling environment.

This is equivalent to requiring that

$$|(1 + E_T)^k(1 + E_R)^k(1 + E_p)^k - 1| < .5 \times 10^{-\mathrm{cp}}.$$

From this we can show that it is sufficient to require that

$$k(|E_T| + |E_R| + |E_p|) < .48 \times 10^{-\mathrm{cp}},$$

since then, denoting $|E_T| + |E_R| + |E_p|$ by E, we have

$$\begin{aligned} &|(1 + E_T)^k(1 + E_R)^k(1 + E_p)^k - 1| \\ &\quad \le e^{kE} - 1 \\ &\quad = kE(1 + kE/2! + (kE)^2/3! + \cdots) \\ &\quad \le kE(e^{kE} - kE/2) \\ &\quad < .48 \times 10^{-\mathrm{cp}}(e^{.048} - .048/2), \end{aligned}$$

this last step being valid because $0 \le kE < .48 \times 10^{-\mathrm{cp}}$, and $\mathrm{cp} \ge 1$. The quantity inside the parentheses is bounded by 1.03, so that the right side of the inequality is bounded by $.5 \times 10^{-\mathrm{cp}}$, as required.

Since $k = 10^{**}t$, we finally obtain the requirement that

$$|E_T| + |E_R| + |E_p| < .48 \times 10^{-\mathrm{cp}-t}.$$

Now, we know that $|E_p| < 5 \times 10^{-p}$, so p would have to be at least as large as $\mathrm{cp} + t + 2$, in which case $|E_p|$ would contribute at most $.05 \times 10^{-\mathrm{cp}-t}$ (i.e., at most .05 to the maximum .48 that we are allowing in the error factor).

In the next section we show that, with $p = \mathrm{cp} + t + 2$, $|E_R| < .10 \times 10^{-\mathrm{cp}-t}$, leaving a final requirement that $|E_T| < (.48 - .05 - .10) \times 10^{-\mathrm{cp}-t} = .33 \times 10^{-\mathrm{cp}-t}$. We show in Section 5 how n can be chosen to ensure that this last requirement is satisfied.

4. THE ACCUMULATION OF ROUND-OFF ERROR

As promised in the preceding section, we now show that $|E_R| < .10 \times 10^{-\mathrm{cp}-t} = 10 \times 10^{-p}$, where E_R is the relative round-off error accumulated in evaluating the truncated Taylor series. The order in which the arithmetic operations are carried out is implied by rewriting the truncated series in the form

$$\left(\left(\cdots\left(\left(\frac{r}{n-1}+1\right)\frac{r}{n-2}+1\right)\frac{r}{n-3}+\cdots+1\right)\frac{r}{2}+1\right)r+1,$$

and noting that $r/(n-2)$, $r/(n-3)$, ..., are calculated in the program *before* multiplying by what is to the left in each case. (If these quotients were not evaluated *before* the corresponding multiplications, the error bound could not be as small as it is.)

The calculations are done in precision $p = \mathrm{cp} + t + 2$. We set $u = 10^{-p}$, and note that the absolute error introduced in the result of any properly rounded arithmetic operation is less than $5u$ if the true result is less than or equal to 10 in magnitude, but is less than $.5u$ if the true result is less than or equal to 1 in magnitude.

We consider first the case when $r > 0$. And, for the time being, we assume that $n - 1 \geq 3$. Then we can show, by induction, that the result obtained as the right parenthesis is reached after each addition of 1 in the above expression (except for the last right parenthesis) is in $[1, 1.5)$ and has an absolute error of less than $9.6u$. It is convenient to denote such a result by $[1, 1.5) + 9.6\epsilon$.

This property is obviously true for the first such right parenthesis, since the floating-point result of evaluating $r/(n-1)+1$ when $n-1 \geq 3$ must be in $[1, 1.5)$ and the absolute error in the division is $< .5u$, while the absolute error in adding 1 is less than $5u$, for a total absolute error of less than $5.5u$, which is less than $9.6u$.

For the inductive step, we note that the result of evaluating r/i, for any $i \geq 3$, must be $(0, 1/3) + .5\epsilon$. (Note that different occurrences of ϵ are not necessarily the same; ϵ merely represents a value which satisfies $|\epsilon| \leq u$.) Then multiplying $[1, 1.5) + 9.6\epsilon$ by $(0, 1/3) + .5\epsilon$ and adding 1, in floating-point arithmetic, produces

$$([1, 1.5) + 9.6\epsilon)((0, \tfrac{1}{3}) + .5\epsilon) + .5\epsilon + 1 + 5\epsilon,$$

where the last $.5\epsilon$ is contributed by the multiplication, and the 5ϵ is contributed by the addition of 1. This result can be rewritten as

$$[1, 1.5) + \tfrac{1}{3}(9.6\epsilon) + 1.5(.5\epsilon) + (9.6\epsilon)(.5\epsilon) + 5.5\epsilon = [1, 1.5) + 9.45\epsilon + 4.8\epsilon^2.$$

But $u < 10^{-3}$ since $p = \mathrm{cp} + t + 2 \geq 3$, so that $4.8\epsilon^2$ can be replaced by $4.8 \times 10^{-3} \times \epsilon = .0048\epsilon$, and the result must be $[1, 1.5) + 9.6\epsilon$, as required.

The result at the last right parenthesis is

$$([1, 1.5) + 9.6\epsilon)(0, \tfrac{1}{2}) + .5\epsilon + 1 + 5\epsilon = [1, 1.75) + 10.3\epsilon.$$

(There is no round-off error in evaluating $r/2$, since r is exact in precision cp and the division is done in precision $p = \mathrm{cp} + t + 2$.)

The final result of evaluating the expression is

$$([1, 1.75) + 10.3\epsilon)(0, 1) + 5\epsilon + 1 + 5\epsilon = [1, 2.75) + 20.3\epsilon.$$

(The error in the product is 5ϵ this time, since the product can exceed 1 for the first time.)

It is easily verified that the cases when $n - 1 < 3$ also produce results that are $[1, 2.75) + 20.3\epsilon$. (These are the cases when there are at most three terms in the truncated series.)

The absolute error incurred is therefore less than 20.3×10^{-p} in magnitude. The relative error E_R is therefore less than 10×10^{-p} in magnitude, as long as $e^r \geq 2.03$, which is the case if $r \geq .71$.

If we now restrict our attention to $0 < r < .71$, and follow through an argument completely analogous to the one just given, we can establish the required result for $r \geq .48$. (We used $[1, 1.32) + 8.08\epsilon$ for the induction.)

Then, considering only $0 < r < .48$, we use an analogous argument once again and finally establish the result for all r in $(0, 1)$. (We used $[1, 1.2) + 7.27\epsilon$ for the induction in this last case.)

For r in $(-1, 0)$, we had to apply the argument only once, since adding 1 in this case can contribute an error of at most $.5\epsilon$. (We used $(2/3, 1] + 2.4\epsilon$ for the induction.)

The required result, namely that the relative round-off error E_R in evaluating the truncated Taylor series satisfies $|E_R| < .10 \times 10^{-cp-t}$, is therefore established for all r in $(-1, 1)$.

5. DETERMINATION OF NUMBER OF TERMS

To determine the number of terms n to be used in evaluating *sum*, we start with the requirement for n that was given at the end of Section 3, namely that n be chosen so that $|E_T| < 33 \times 10^{-p}$, where E_T is the relative truncation error in using the truncated series approximation to e^r.

An ideal result would be the smallest integer n that satisfies this inequality. We will develop a method for finding, quite quickly, a value of n that is only slightly larger than this ideal value.

First note that

$$e^r E_T = \left(\frac{r^n}{n!} + \frac{r^{n+1}}{(n+1)!} + \frac{r^{n+2}}{(n+2)!} + \cdots\right),$$

so that

$$|E_T| \leq \frac{|r|^n e^{-r}}{n!}\left(1 + \frac{|r|}{n+1} + \frac{|r|^2}{(n+1)^2} + \cdots\right) = \frac{|r|^n e^{-r}(n+1)}{n!(n+1-|r|)}.$$

It is therefore sufficient to require n to satisfy

$$\frac{|r|^n e^{-r}(n+1)}{n!(n+1-|r|)} < 33 \times 10^{-p}.$$

If we now use Stirling's formula,

$$n! = \sqrt{2\pi}\, n^{n+1/2} e^{-n+\theta/12n}, \; n > 0, \; 0 < \theta < 1,$$

(see Davis [8, p. 257]) to replace $n!$ in the above inequality, and take logarithms of each side, we can rearrange terms to obtain

$$n \ln n - n \ln |r| - n - p \ln 10 + c \geq 0,$$

where c satisfies

$$c < \ln 33 + \frac{1}{2}\ln 2\pi + \frac{\theta}{2n} + \ln\frac{\sqrt{n}(n+1-|r|)}{n+1} + r.$$

A lower bound for the right side of this inequality is obtained by setting $\theta = 0$, $n = 1$, and $r = -1$. Rounding the individual terms appropriately leads to the following sufficient condition for c:

$$c < 3.4965 + .9189 + 0 - .6932 - 1 = 2.7222.$$

It is therefore sufficient to have n satisfy

$$g(n) \geq 0,$$

where

$$\begin{aligned} g(n) &= n \ln n - n \ln |r| - n - p \ln 10 + 2.7222 \\ &= n \ln\left(\frac{n}{e\,|r|}\right) - p \ln 10 + 2.7222. \end{aligned}$$

For fixed values of r and p, $g(n)$ is an increasing function of $n (n > 0$, of course), so any upper bound for the solution of $g(n) = 0$ will satisfy the required inequality $g(n) \geq 0$. Morevoer,

$$g'(n) = \ln\left(\frac{n}{|r|}\right)$$

is also an increasing function of n, so the function $g(n)$ is concave upward, and one Newton step in an attempt to solve the equation $g(n) = 0$ will provide an upper bound for the solution of $g(n) = 0$.

The Newton iteration is easily shown to be

$$\begin{aligned} n_{i+1} &= \frac{n_i + p \ln 10 - 2.7222}{\ln(n_i/|r|)} \\ &= \frac{n_i \log e + p - 2.7222 \log e}{\log(n_i/|r|)}, \end{aligned}$$

and our plan is to take only one step, after setting $n_0 = p$, and to make sure that all approximations made in the course of this calculation are directed so that the final result is guaranteed to be an upper bound.

Setting $n_0 = p$ in the numerator, and noting that $\log e + 1 < 1.435$ while $2.7222 \log e > 1.182$, we determine that $1.435p - 1.182$ is an upper bound for the numerator. In the precision specified in the program, namely $\max(p, 4)$, it is a straightforward matter to verify that this bound is evaluated exactly.

The denominator is more troublesome. With $n_0 = p$, it becomes $\log(p/|r|)$. We first obtain a lower bound for $p/|r|$. This is done with the *divrd* function in Numerical Turing, and the result is denoted by *pbyr* in the program. (Note that neither overflow nor zerodivide can occur, since $|r| > .9 \times 10^{-cp}$.) Then the fraction part of *pbyr* is denoted by f, which is normalized so that $.1 \leq f < 1$.

A lower bound for the denominator is then

$$\log(pbyr) = \text{getexp}(pbyr) + \log f.$$

The remaining difficulty with the denominator is to compute a lower bound for $\log f$, where $.1 \leq f < 1$. Such a bound can be obtained from the rational approximation

$$\frac{f-1}{a(f-.1)+.9}$$

to log f, where a is chosen so that the derivative of this expression is 1/ln 10 when $f = 1$. This leads to $a = \ln 10/.9 - 1 = 1.558427\ldots$. The resulting approximation is exactly equal to log f when $f = .1$ and when $f = 1$, and the choice of a ensures that its derivative equals the derivative of log f at $f = 1$. It can be verified that this expression is a lower bound for log f when $.1 \le f < 1$.

Since $\log f < 0$ when $.1 \le f < 1$, we change the sign of this expression and consider

$$\frac{1-f}{a \times f + b},$$

where $b = -.1 \times a + .9 = .7441572\ldots$. This expression is positive when $.1 \le f < 1$, and we require an upper bound for it, so that our approximation to log(*pbyr*) will be a lower bound.

An upper bound for this expression is

$$\frac{1-f}{1.558 \times f + .7441},$$

since 1.558 and .7441 are lower bounds for a and b, respectively.

In the precision specified in the program, $1 - f$ is evaluated exactly. Directed roundings are then used in a straightforward way to produce a lower bound for log(*pbyr*), and hence a lower bound for the denominator in the expression for n.

A final *divru* applied to the numerator and denominator, followed by taking the ceiling of this result, provides the required value of n.

As a check on how close the value of n resulting from these calculations is to the ideal value, we compared the results obtained in this way with the smallest value of n needed to satisfy

$$\frac{|r|^n(n+1)}{n!(n+1-|r|)} < 33 \times 10^{-p}.$$

The left side of this inequality is the bound we used for $|E_T|$, except that the factor e^{-r} has been omitted. The smallest value of n satisfying this inequality will be close to the ideal value. (For all but very small values of n, it cannot differ from the ideal value by more than 1.)

We made the comparisons for $r = 10^{-10}$, .01, .1, .3, .5, .7, and .999. For precisions less than 50, the value of n determined by the program was usually 1 or 2 more than necessary to satisfy the above inequality. (In fact, at precision 49 the values were exactly 1, 2, 2, 2, 1, 1, 2, and 2, respectively, more than necessary for the above values of r, for which the smallest values satisfying the inequality were 5, 17, 24, 30, 34, 37, 39, and 40, respectively.) For precisions up to 100, the value determined by the program was never more than 4 more than necessary.

Now that the method for calculating n has been explained, we can show that there is no possibility of underflow in the evaluation of *sum*. We first show that

$n < 4p$. This follows from the expression on which the evaluation of n is based, namely

$$\frac{1.435p - 1.182}{\log(p/|r|},$$

and noting that $p \geq 3$, while $|r| < 1$. Not only is the exact value of this expression bounded above by

$$\frac{1.435p - 1.182}{\log 3},$$

but so is the upper bound for this expression that is calculated by the program. Since $\log 3 = .4771\ldots$, n must be less than $4p$.

For underflow to occur in the evaluation of *sum*, the calculation of $r/(n - 1)$ would have to underflow. Since $n < 4p$, r would then have to satisfy

$$|r| < 4p \times 10^{-10p-1},$$

since the smallest nonzero magnitude in precision p in Numerical Turing is $.1 \times 10^{-10p}$. But at this stage in the program, $|r| > .9 \times 10^{-cp}$, so underflow cannot occur. (In fact this restriction on $|r|$ is not needed to show that $r/(n - 1)$ cannot underflow, but the proof is more complicated if this restriction is not used.)

6. TESTING

The accuracy of the program *exp* can be tested in a straightforward manner. To test the value returned by *exp* for a particular value of its argument x in a particular precision cp, the basic idea is to use higher precision to calculate a very accurate approximation to e^x in the form of a small interval, which is easily guaranteed to contain e^x, and then to check that all points within this interval differ from the approximation provided by *exp* by less than 1 ulp.

A lower bound for the true value of $e^{|x|}$ is given by

$$1 + |x| + \frac{|x|^2}{2!} + \cdots + \frac{|x|^{n-1}}{(n-1)!},$$

for any value of n. An upper bound is given by

$$1 + |x| + \frac{|x|^2}{2!} + \cdots + \frac{|x|^n}{n!}\left(1 + \frac{|x|}{n+1} + \frac{|x|^2}{(n+1)^2} + \cdots\right)$$
$$= 1 + |x| + \frac{|x|^2}{2!} + \cdots + \frac{|x|^{n-1}}{(n-1)!} + \frac{|x|^n(n+1)}{n!(n+1-|x|)},$$

provided $n + 1 > |x|$. For the bounds obtained by our test program, the appropriate directed roundings are used with each floating-point operation, and the summing is continued until $n + 1 > |x|$ *and* the relative difference between the two sums is less than a quantity that is small compared to 10^{-cp}.

For $x > 0$ these two bounds provide the interval referred to earlier, while for $x < 0$ their reciprocals, appropriately rounded and of course interchanged, provide the required interval.

The test program checks only sufficient conditions which ensure the correctness of *exp*, and it may be necessary in some cases (e.g., in rare cases where part of the small interval differs by an ulp or more from the approximation provided by *exp*, or when cp or $|x|$ are relatively large) to adjust these conditions. For this reason the "higher precision" in which the calculations are done and the "small quantity" used to bound the relative difference between the two sums are parameters to the test program. Usually we chose them to be cp + 10 and 10^{-cp-2}, respectively. (The latter ensures that the two sums differ by less than 1 unit in the (cp + 2)th place.)

The accuracy of *exp* was easily confirmed for a large number of argument values, including many near the overflow and underflow thresholds, and for a variety of precisions, ranging from precision 1 up to precision 100.

7. CONCLUDING REMARKS

We have shown theoretically that the function *exp* in Figure 1 returns a result that differs from e^x by less than 1 unit in the last place, for any p-digit argument x (any $p > 0$), provided x is not too close to values for which e^x would overflow or underflow.

The program is also reasonably efficient, considering that it is a "variable precision" program. It uses one extra arithmetic operation per term, compared to a fixed-length polynomial such as a Chebyshev polynomial. And, for any particular precision, it may require more terms than the corresponding Chebyshev polynomial. But there is some compensation for smaller values of $|r|$, since the calculations involved in determining n take advantage of the smallness of $|r|$, and the number of terms can be fewer than required by the corresponding Chebyshev polynominal.

On the other hand, it should be acknowledged that the Arithmetic-Geometric Mean method will be more efficient asymptotically for large precisions (probably >100); see Brent [3].

We have drawn attention to the usefulness of three language extensions, namely precision control, directed roundings, and the *getexp* and *setexp* functions. These extensions have made it possible to program in a straightforward way exactly what we wished to implement. The resulting program is relatively easy to follow (although, admittedly, the readability of the program would be improved if the built-in functions for directed roundings were replaced by infix operators—as is planned for a future version of Numerical Turing).

We also claim that the language features make it easier to prove the program correct. We mean this in the sense that the proof is not made more complicated because of limitations imposed by the programming language, or by any irregularities in the floating-point arithmetic. The difficulties in the proof are inherent in the algorithm itself.

The same three language extensions have also made it possible to write, again in a straightforward way, a test program which can test *directly* whether or not the program *exp* meets its accuracy requirement. This is in contrast to the *indirect* testing one usually has to resort to. (See Cody and Waite [7] for a very thorough treatment of this topic.)

Precision cannot exceed 200 in current implementations of Numerical Turing. As a result, the program *exp*, as it stands, will fail at the point where the precision is declared to be p if *currentprecision* $+ t + 2 > 200$. For this reason we would in practice replace the expression *currentprecision* $+ t + 2$ used in the definition of p with min(*currentprecision* $+ t + 2$, *maxprecision*). The function *maxprecision* returns the value 200 in the current implementation of Numerical Turing.

The program will then no longer fail at the point at which the precision is declared to be p, but the program's specification would have to be modified. The error will still be less than 1 unit, but it will not be in the last place if $p > 200$, but rather in the place which is d digits back from the last place, where $d = p - 200$.

The program will also fail if $|x| > 23 \times$ *currentprecision*. The current implementation of Numerical Turing replaces the corresponding *if–end if* construct with a *pre* statement [9] which causes the failure to take place at the point of invocation of, rather than inside, the *exp* function. We leave *exp* in its present form because Numerical Turing will soon have facilities for raising and handling exceptions, and the *assert false* statement can then be replaced by a statement that causes an exception to be raised at the point of invocation.

REFERENCES

1. ABRHAM, A. Variable precision elementary functions. M.Sc. thesis, Dept. of Computer Science, Univ. of Toronto, Toronto, 1985.
2. BORWEIN, J. M., AND BORWEIN, P. B. The Arithmetic-Geometric Mean and fast computation of elementary functions. *SIAM Rev. 26*, 3 (July 1984), 351–366.
3. BRENT, R. P. Fast multiple-precision evaluation of elementary functions. *J. ACM 23*, 2 (Apr. 1976), 242–251.
4. BRENT, R. P. A FORTRAN multiple-precision arithmetic package. *ACM Trans. Math. Softw. 4*, 1 (Mar. 1978), 57–70.
5. BRENT, R. P. Unrestricted algorithms for elementary and special functions. In *Proceedings of the IFIP Congress 80* (Tokyo and Melbourne, Oct. 1980), Simon Lavington Ed., North-Holland, Amsterdam, 613–619.
6. CLENSHAW, C. W., AND OLVER, F. W. J. An unrestricted algorithm for the exponential function. *SIAM J. Numer. Anal. 17*, 2 (1980), 310–331.
7. CODY, W. J., AND WAITE, W. *Software Manual for the Elementary Functions.* Prentice-Hall, Englewood Cliffs, N.J., 1980.
8. DAVIS, P. J. Gamma function and related functions. In *Handbook of Mathematical Functions*, M. Abramowitz and I. A. Stegun, Eds., National Bureau of Standards, Applied Mathematics Series 55, U.S. Government Printing Office, Washington D.C., June 1964, 253–293.
9. HOLT, R. C., AND CORDY, J. R. The Turing language report. Tech. Rep. CSRI-153, Dept. of Computer Science, Univ. of Toronto (revised July 1985). (An earlier version of this report also appears as an appendix in a text by Holt, R. C. and Hume, J. N. P., *An Introduction to Computer Science Using the Turing Programming Language.* Reston, Reston, Va., 1984.)
10. HULL, T. E. Precision control, exception handling and the choice of numerical algorithms. In *Proceedings of the Dundee Conference on Numerical Analysis*, G. A. Watson, Ed., Springer-Verlag, New York, 1982, 169–178.
11. HULL, T. E. The use of controlled precision. In *Proceedings of the IFIP TC2 Working Conference on the Relationship between Numerical Computation and Programming Languages* (Boulder, Colo., Aug. 1981), J. K. Reid, Ed., North-Holland, Amsterdam, 1982, 71–84.
12. HULL, T. E., AND ABRHAM, A. Properly rounded variable precision square root. *ACM Trans. Math. Softw. 11*, 3 (Sept. 1985), 229–237.
13. HULL, T. E., ABRHAM, A., COHEN, M.S., CURLEY, A. F. X., HALL, C. B., PENNY, D. A., AND SAWCHUK, J. T. M. Numerical Turing. *ACM SIGNUM Newsl. 20*, 3 (July 1985), 26–34.

Received January 1986; accepted July 1986.